

16k UTILITY EPROM
BBC MODEL B



The SLAVE DRIVERS GUIDE

SLAVE DRIVERS GUIDE

by

J. AUGHTON and I. PIUMARTA

JANUARY, 1985

To: TRISH and KAREN

SLAVE

A&F SOFTWARE LTD UTILITY ROM FOR THE BBC MICRO MODEL B

"SLAVE" IS A TRADEMARK OF A&F SOFTWARE LTD

Copyright © 1984 A&F Software Ltd. All worldwide rights reserved. No part of this manual may be reproduced by any means without the prior written consent of the copyright holders.

The accompanying program "SLAVE", which is supplied in Eprom, is subject to copyright. No part of "SLAVE" shall be copied or reproduced for any purpose.

A&F SOFTWARE LTD

**UNIT 8, CANAL SIDE INDUSTRIAL ESTATE, WOODBINE STREET EAST,
ROCHDALE, LANCs. Tel: (0706) 341111**

CONTENTS

	Page		Page
Introduction	5	*CHAR	26
Compatibility	5	*EDIT	27
Fitting SLAVE	6	Machine code and disc utilities	
*HELP Commands	7	*MOVE	31
The format of SLAVE commands	10	*CROM	32
The *FX 30 command	11	*EDKEY	35
The 'No Language' environment	12	*SLOW	36
		*GET	37
		*PHEX	38
		*GO	38
BASIC Utilities		Memory editing	39
*BAD	14	*HEX	41
*EXTEND	15	*MON	43
*PACK	16	*SPACE	45
*SORT	18	*DGET	46
*PAGE	19	*DISCED	47
*ENVELOPE	20	*DIS	50
*FIND	21	*DEBUG	53
*REPLACE	22	*CRC	59
*XREF	23	*REL	61
*TRON	24	Error messages	62
*TROFF	25	Summary of commands	65

SLAVE DRIVERS GUIDE

INTRODUCTION

SLAVE is a 16k program providing the BBC computer user with a large selection of powerful utilities on a single ROM.

The design philosophy of **SLAVE** has been to provide genuinely useful routines that can be relied upon to work, and that will provide a comprehensive development and debugging aid. Despite its power, **SLAVE** is easy to use; this being a prime consideration in its design. You do NOT need to be a computer expert to use **SLAVE**; however, through its use you will certainly learn more about your computer and develop your skills as a programmer.

The scope and depth of **SLAVE's** many routines should suit even the most demanding user and each of the utilities has been included because it has been found to be of practical value - there are no cosmetic routines in **SLAVE**! Some of its facilities can be found on other ROM chips and, where this is the case, the **SLAVE** routine should be at least as useful and usable as its counterpart.

Before you use **SLAVE**, read through this manual carefully to familiarise yourself with the utilities and the ways in which they may be applied.

COMPATIBILITY

SLAVE is compatible with BASICs 1 and 2, OS 1.00 and 1.20, and with a large number of sideways ROMs, language ROMs, and games that it was possible to test it with.

The MCODE section of **SLAVE** is compatible with second processors. All references then made to RAM (MON, HEX, etc.) will occur within the host processor. No BASIC utilities will function as BASIC programs are held in the parasite processor's memory.

Any conflicts with other ROMs are avoided by the use of a 'Z' prefix to any **SLAVE** command.

Any **SLAVE** command may be used within a BASIC program. If there are errors generated, they may be trapped with the normal 'ON ERROR' coding.

If the computer is switched on with no BASIC ROM present, the message 'Language?' will be displayed and the beeper will sound. The no-language environment may be entered with Z—BREAK.

The disc utilities will work with any Acorn format, 40 or 80 track, single density disc system.

FITTING SLAVE

UNLESS YOU ARE CONFIDENT ABOUT INSTALLING A NEW ROM YOU SHOULD READ THESE INSTRUCTIONS CAREFULLY BEFORE REMOVING YOUR SLAVE FROM ITS PROTECTIVE PACKAGING

ENSURE THAT THE COMPUTER IS DISCONNECTED FROM THE MAINS SUPPLY

First locate the four screws which hold the two halves of the computer case together. Two will be found on the underside of the case near the front, and two on the rear panel at the edges. Each screw should be labelled 'FIX>'. Carefully undo these screws (bearing in mind that the upper half of the case will become detached when the fourth is removed) and place the computer on a flat, stable surface.

Raise the front edge of the lid a little way and then lift the whole lid off vertically. Put the lid to one side. Tilt the computer from the front to reveal the two (three on old machines) keyboard retention bolts. These bolts should be loosened with a screwdriver. Lower the computer and remove the keyboard retention nuts by hand. The keyboard should now be lifted vertically a little way and turned clockwise to reveal the four sideways ROM sockets. There may be fewer than four empty sockets depending on whether your computer has any extra ROMs fitted.

Carefully remove your **SLAVE** from its protective packaging. You should take great care not to touch any of the metal pins as even a slight amount of static electricity could damage it. Locate the small notch in the top surface of the ROM. When installed, this notch should be facing the rear panel of the computer. Now insert the ROM into the right-most socket, taking care to avoid bending any of the pins. You should be especially aware of the danger of bending any of the pins inwards under the body of the ROM itself, as this can be very difficult to detect at a later stage. Double check that **SLAVE** is installed the right way round - an incorrectly oriented ROM will almost certainly be destroyed if the power is switched on.

Replace the keyboard retention nuts, finger tight. Replace the lid of the computer taking care not to bend any of the three status lights on the left side of the keyboard. Plug in and switch on.

Typing '*HELP' should produce a number of ROM titles, including '**SLAVE 1.00**'. If it does not switch off immediately, un-plug, and re-check the installation. If no fault can be found, carefully remove the ROM, return it to its protective packaging, and seek professional advice. If all is well, tighten the keyboard retention bolts with a screwdriver and secure the upper half of the case.

*** HELP COMMANDS**

***HELP**

This command will produce the usual list of ROM titles, together with any associated keywords. When you have fitted **SLAVE** into your computer you will see something like this:

SLAVE 1.00

PROG

MCODE

OS 1.20

***HELP PROG**

'PROG' is one of **SLAVE's** keywords. This command will list the BASIC utilities showing their correct syntax. The list should read:

SLAVE BASIC

PROG

BAD (char)

EXTEND (char)

PACK (byte)

SORT array (no) (start)

PAGE (page) (R)

ENVELOPE	(no)
FIND	name/kywrd (no)
REPLACE	name1 name2
XREF	(N)
TRON	(lno)
TROFF	
CHAR	(first) (last)
EDIT	(lno)

Typing '*PROG' by itself will clear the screen before displaying the list.

***HELP MCODE**

This command will produce a listing of all **SLAVE's** machine code and disc utilities. The list should read as follows:

SLAVE MONITOR

MCODE

MOVE	first + len/last to
CROM	(no/rsp)
EDKEY	(first) (last)
SLOW	(dly)
GET	string/'hex'
PHEx	first + len/last
GO	addr
MON	(addr)
HEX	(addr)
SPACE	(drv)
DGET	string/'hex'
DISCED	(drv)
DIS	(first) (+ len/last) (sw)
DEBUG	(addr)
CRC	fsp first + len/last
REL	first + len/last to

Typing '*MCODE' by itself will clear the screen before displaying the list.

***HELP command**

Where 'command' is any of **SLAVE's** keywords will display the syntax for that command alone.

In the lists above, this notation has been used:

- (. . .) — Brackets indicate optional parameters
- R, N — Capitals denote literal parameters
The remaining parameter names refer to the type of information required and should not be entered literally.
- char — any printable character
- byte — a decimal number in the range 0 — 255
- array — an array of one dimension
- start — the position of the key field in a string
- page — a new value for PAGE
- name — a BASIC variable
- kywrđ — a keyword recognised by BASIC
- lnđ — a BASIC line number
- first — the address of the first byte in a block OR item in a list
- len — the length of a section of memory
- last — the address of the last byte in a block OR item in a list
- to — the destination address of a block of memory
- rsp — a ROM specification (name)
- dly — a value for a delay
- string — a list of printable characters
- 'hex' — hexadecimal bytes surrounded by single quote marks
- addr — an address in hexadecimal
- drv — a (disc) drive number
- sw — a hexadecimal switch byte
- fsp — a file specification

The ranges of values for the above parameters are explained more fully in the individual descriptions of commands.

THE FORMAT OF SLAVE COMMANDS

Each **SLAVE** command must be preceded by a '*' character, so that the Operating System can pass the command to **SLAVE** for processing. The basic format of the line is given in the section on 'HELP' but it is not necessary to adhere rigorously to that format provided certain rules are obeyed.

The keyword itself may be entered in upper or lower case, or a mixture of the two and can be terminated at any point with a '.' character. **SLAVE** will select the first routine that matches the keyword up to the '.'. Thus, *ENVE. or even *EN. will call the envelope routine, but *PA. will call PACK not PAGE, because it has a higher priority on the HELP page.

To avoid clashes with other ROMs that allow '*' commands, any **SLAVE** command can be prefixed by an optional 'Z' character. For example, *ZMove is equivalent to *MOVE as far as **SLAVE** is concerned, whereas *MOVE itself might be picked up by another chip in a higher priority socket. The 'Z' prefix is a tag that identifies the command to **SLAVE**.

The parameters to a keyword must follow in the order specified, and must be separated from each other by one or more spaces. Unless the keyword is terminated with a '.', there must be at least one space between it and the first parameter.

Any numeric parameter can be replaced by one of the resident integer variables, A% — Z%. In this case the two least significant bytes that make up the integer will be used in place of the parameter.

For example, the commands below are all equivalent (assuming that A% has previously been set to 150)

*SLOW	150
*SL.150	
*ZsLoW	150
*sl. A%	
*Slow	A%
*zSlo.A%	

THE *FX 30 COMMAND

SLAVE will always respond to a *FX 30 command, which will be used to condition certain internal flags. The two least significant bits of the second parameter determine the action to be taken as shown in this table:

Binary	Decimal	Action
0 0	0	Enable SLAVE . Colours on.
0 1	1	Enable SLAVE . Colours off.
1 0	2	Disable SLAVE . (Colours on).
1 1	3	Disable SLAVE . (Colours off).

Notice that the second parameter defaults to zero so that *FX 30 is the same as *FX 30 0.

Once **SLAVE** has been disabled it will only process the *FX 30 command and will not even respond to *HELP. Pressing 'BREAK' will not alter the status of **SLAVE**, but a hard reset (CTRL—BREAK) will restore it to its default state, ie. enabled with colours on.

The disable feature is 'soft' in that **SLAVE** may be re-enabled by a further *FX 30 call. If it is required to turn off **SLAVE** so that NO commands are processed, then a 'hard' disable is available by setting the top bit of the *FX 30 parameter. Thus, *FX 30 n where $n > 127$ will inform the Operating System that **SLAVE** is not to be called again until 'BREAK' is pressed. Some machine code game programs that (ab)use Operating System workspace may not function correctly until **SLAVE** is fully disabled in this way.

SLAVE's HEX, MON and DEBUG routines all make extensive use of colour in their display screens. This is not always readable on some TV sets and monitors, so the 'colours off' feature has been included. Once the colours have been switched off these screens will appear in black and white with the highlighted fields being identified by '—>' characters.

THE 'NO LANGUAGE' ENVIRONMENT

If Z—BREAK is pressed then the No Language environment is entered - this is similar to the native environment on second processors without BASIC. An asterisk prompt is displayed at which a command line may be entered for immediate evaluation by the operating system. All operating system calls available from BASIC may be used, and the leading asterisk in such commands may be dropped.

This environment is entered with the Operating System High Water Mark (OSHW) set to its correct value (i.e. on disc machines it will be at least &1900, cassette machines &E00), but with no ROMs of a lower priority than **SLAVE** initialised. This means that on a disc machine with **SLAVE** in the right-most socket, all file I/O will default to the tape filing system. To select the disc filing system, the command 'DISC' must be issued. Due to the position of BASIC in most machines, a crashed program moved to &E00 on a disc machine can be interrupted via a Z—BREAK without corrupting locations &1900 and &1901.

If Z—BREAK is pressed and a BASIC ROM exists in the machine, BASIC can be re-entered either with another BREAK or by typing the command 'BASIC'.

BASIC utilities

***BAD (char)**

Recover 'Bad' program

This command cures the BASIC 'Bad Program' error. Before BASIC can RUN or LIST a program, it checks to see if the program conforms to certain standards and, if not, generates this error. Unfortunately 'Bad program' is a difficult error to overcome as it usually means that the program has been corrupted in some way. *BAD will repair ANY program to the point where it is listable and, providing the corruption is not too severe, you should be able to edit the program back to health.

There are various causes for the error but perhaps the simplest way to generate it is to type $?(PAGE + 3) = 0$ when you have a program in memory. In this particular case **SLAVE** will be able to recover all of the program.

Another form of 'Bad' program occurs when a BASIC program fails to load properly from tape (or, very rarely, from disc). The program can be forced to load by using *OPT 2,0 (as described in the User Guide) even though 'Block' and 'Data' errors may occur. *BAD will very often recover the program sufficiently for you to be able to piece it back together again.

It may be in the nature of the corruption that certain control codes (characters with ASCII codes less than 32) have appeared in your program, in which case any attempt to LIST it may fail or even appear to hang up the machine. This is actually a standard trick used in some programs to render them 'unlistable'.

To see this for yourself add the line OREM* to any BASIC program and then type $?(PAGE + 5) = 21$. Your program will not list beyond the word 'REM' because the control code you have added to the program turns off the VDU drivers so that no further screen output occurs. Despite this, the program will still run as normal.

The optional parameter that may be used with *BAD can be any printable character that will be used to overwrite any control codes in the program, thus highlighting them and making the program listable.

Once a program has been recovered it may be necessary to RENUMBER it if the line numbers have been affected, and to edit out any control codes that have been highlighted by being replaced by 'char'.

If a program is long, *BAD may not recover all of it, as the routine may well find a logical end to the program that is not really its true end. It is for you to decide if the program is to be extended and this is the function of the next **SLAVE** command.

***EXTEND (char)**

Extend BASIC program.

Starting from the current BASIC program stored at PAGE, this command will add at least one (and possibly very many) lines to the end of it, thereby extending the program. *EXTEND works by linking the last line of your program to whatever lies directly after it in memory - this may or may not make sense, but you will be left with a longer, yet still valid, program that may be edited or RENUMBERed.

To see the command in action, load a long program (50 lines or more) into store and then type NEW followed by:

```
10 REM NOW IT'S GONE!!
```

Obviously, LIST will just give the above line but a *EXTEND command will recover nearly all of the original program, except for the first line or so as these have been overwritten by a new line 10. RENUMBER and a bit of judicious editing will now enable you to get the original program back.

The optional 'char' parameter is used in exactly the same way as that in *BAD, that is it will overwrite control codes that may appear as a result of the recovery process.

***PACK (byte)**

Compact BASIC program

During the development of long BASIC programs, it is quite easy to run out of usable memory resulting in 'Bad Mode' or 'No Room' errors, and the only solution is to reduce the size of the program. The PACK facility will remove unnecessary bytes from a program without affecting the integrity of the program in any way.

A variety of packing features are available and they are selected by the setting of 'byte' which is a decimal number in the range 0 — 255. Each of the pack facilities has an associated number:

Number	Effect
0	Remove spaces from the ends of lines only.
1	Remove spaces from the starts of lines only.
2	Remove all REMs.
4	Remove assembler comments.
8	Remove all unnecessary spaces.

Several effects can be used simultaneously if 'byte' is set suitably; for example, to remove REMs and assembler comments only, use 'byte' = $2 + 4 = 6$.

Notice that *Pack will ALWAYS remove spaces from the ends of BASIC lines - these are often included inadvertently when the program is being typed-in or when lines are edited using the copy' key. *PACK is equivalent to *PACK 0 in that it will only remove these spaces.

No lines are removed from a program. If a line consists of a REM statement only, then the REM is deleted, but the line number is retained (what is actually left is a line of length 4 containing no text — this is the shortest possible line acceptable to BASIC).

Assembler comments are recognised by their being preceded by a '/' character. Bytes up to the next ':' are removed, as assembler comments may be freely mixed with actual instructions (unlike REMs in BASIC); only the comments are removed.

When the *PACK 8 option is used to remove spaces, only unwanted spaces are deleted - certain spaces are retained in REM lines, command lines (those preceded by '*'), IF statements, and within quote marks.

For example:

```
10 REM AN EXAMPLE OF *PACK
20C=0: FOR I= 1TO 10
30 IF I=3 C= 4
40 NEXT I
50 * HEX 8000
60PRINT "END OF DEMONSTRATION"
```

followed by *PACK 8 will produce:

```
10REM AN EXAMPLE OF*PACK
20C=0:FORI=1TO10
30IFI=3 C=4
40NEXTI
50*HEX 8000
60PRINT"END OF DEMONSTRATION"
```

Similarly, the effect of *PACK 3 on the original program would be to produce:

```
10
20C=0: FOR I= 1 TO 10
30IF I=3 C=4
40NEXT I
50* HEX 8000
60PRINT "END OF DEMONSTRATION"
```

***SORT array (no) (start)**

Sort specified array into ascending order.

This command can be used on any integer, real or string array of one dimension to sort the contents into ascending order by value (for integers and reals) or by ASCII code (for strings).

If only 'array' is specified, then the entire contents of the array are sorted including the zeroth element. Thus if you define: DIM fred%(30) in your program, *SORT fred% will sort 31 integers, even if you have not assigned values to them all. In this case unassigned elements will be taken to have value 0. In the case of strings, each element HAS to be assigned, otherwise you will get a 'String too short' error.

The second parameter, 'no' (a decimal number) is used to restrict the sort to the first few members of the array so as to avoid sorting invalid or unassigned elements. For example: *SORT jim 9 will sort the contents of jim(0) to jim(9) into ascending order - all other members of the array will be untouched.

When sorting string arrays the third parameter may be used to define a key for the sort. 'start' is a decimal number in the range 1 - (length of shortest string) and determines the point in the string at which the sort routine will start comparing strings. Any characters before this position are completely ignored. The command:

***SORT sheila\$ 300 3**

will sort the first 301 elements of array sheila\$ into alphabetical order starting at the third character of each string. If any string is less than three characters long, a 'String too short' error will occur.

This feature is useful for sorting records on a file where each record consists of a string containing a number of fields. Suppose Phonedata\$(100) contained strings in this format:

Characters 1—5	Telephone number
Characters 6—20	Surname
Characters 21—60	Address

To sort the file into telephone number order, it should be read into array Phonedata\$ and followed with: *SORT Phonedata\$ 100. In this case the sort will start comparing strings from the first element (in other words, the default value of 'start' is 1). However to sort the array into surname order,

we can use the command *SORT Phonedata\$ 100 6 and similarly, *SORT Phonedata\$ 100 21 will sort the array into address order.

When using the string sort facility, bear in mind that the zeroth element of the array will be sorted and that it must be defined. If this is not convenient, then define it yourself as (for example) Phonedata\$(0) = "zzzzz" — this will ensure that it is sorted to the very end of your array so that it can subsequently be avoided.

***PAGE (page) (R)**

Move BASIC program to new address.

The BASIC variable PAGE defines the start of the area available to a program. By altering PAGE, it is possible to load a BASIC program into a new place in store without corrupting the existing contents of the machine. This is a very useful facility but it can be difficult to organise unless great care is used. For example, all programs will want to store their variables at TOP, and unless TOP is recalculated for each program before it is run, it is quite likely that some memory will be corrupted somewhere and at least one program will be destroyed.

On BREAK or power-up, PAGE is set to the 'Operating System High Water Mark' (OSHW, for short) which is determined by the ROMs resident in your computer. For tape-based machines, the usual value is &E00, and for machines fitted with a disc interface, it will probably be &1900 - check by typing 'PRINT' ~PAGE' following switch-on.

*PAGE on its own will take the BASIC program stored at the current value of PAGE and move it to OSHWM - PAGE is reset to OSHWM and TOP is recalculated. If there is no valid BASIC program at PAGE, a 'Bad program' error will occur.

The optional parameter 'page' should be an address (in hex) that you wish to move your BASIC program to. PAGE must lie on a page boundary (that is, it is of the form &xx00) and strictly speaking, 'page' should be of this form too. However **SLAVE** will accept a value such as 1234, in which case, PAGE will be reset to &1200.

The program may be moved up or down in RAM — even to an area that overlaps itself. The only restrictions are that the program must not be moved so high up the memory that it (or its variable stack) overflows onto the screen or so low that vital system information is overwritten.

If the optional character 'R' is added to the command line then the program will be run as soon as it has been moved. This feature allows long disc-based programs to be loaded, relocated and then run from the new address with a single command. This particular sequence of events is very common when running BASIC programs from disc and can otherwise be difficult to control. The full format *PAGE command has been designed to make the process much easier.

***ENVELOPE (no)**

Display envelope definitions.

If no parameter is used (ie. by typing *ENVELOPE only) the current definitions for envelopes 1 to 4 are displayed in their BASIC formats. These definitions may be copied from the screen and are consistent with the syntax as defined in the User Guide (for example in their use of negative numbers as parameters). Also displayed is the message 'Do 5—16?' and any response other than 'Y' will cause the routine to exit. A 'Y' response will display definitions for envelopes 5 through 16, although these may not (all) have been defined explicitly. The storage area for these definitions is shared by certain buffers and so the results may not necessarily be meaningful (except that they are all valid envelopes and may be copied from the screen and used freely with SOUND commands).

One use of this feature is to find out how certain sound effects are achieved by a commercial game program - sometimes the envelopes are defined in machine code routines and it is not easy to see what definitions have been used.

An alternative syntax is to use *ENVELOPE followed by a decimal number in the range 1 — 16. In this case only the definition for that envelope will be listed.

***FIND name/keywrd (no)**

Find references to BASIC variable 'name' or keyword

This command will list all the lines of the current BASIC program containing the variable 'name'. The use of *FIND is limited to true variable names (eg. Jeff%), constants (eg 33005) and BASIC keywords (eg. PRINT). Assembler mnemonics will also be found as BASIC treats them like variable names.

This feature is actually far more useful than the usual 'string search' found with other utilities because each line is studied very closely for the variable (for example, nearly all string searches will find a 'C' in the line 400 GOTO 1000 — the is potentially disastrous if used in conjunction with a 'Replace' routing). *FIND will not locate the search variable within quotes or a REM statement, because BASIC would not either.

The optional parameter 'no' is a decimal number in the range 0 - 127 and determines how many lines are listed following the line at which the variable is first located. This is useful for searching for and listing both functions and procedures by using the command *FIND DEF 10. In this method of use, the sets of lines are separated by a single blank line to make the display easy to follow.

Once the variable has been located the line(s) is (are) listed and the routine waits for you to press 'RETURN' before proceeding with the search. You may quit any time by pressing 'ESCAPE'.

Examples:

Valid: *FIND fred
 *fi.Array\$ 0
 *FIN. Broughton 7
 *FIND DEF 12
 *FIND TAB (2

(The last example shows that the keyword TAB also includes a '(' character as part of its definition. The section entitled 'Minimum Abbreviations' in the User Guide shows the full definition of each keyword)

Invalid: *Find My Word
 *FIND Ian 200
 *Fi. PRINTER
 *FIN. INSTR

(The last example is actually valid, but it won't find what you might expect -see the explanation for TAB().

The listed lines produced by the FIND facility take account of the setting of 'LISTO 1'.

Neither this nor the following routine can find line numbers used in (for example) RESTORE statements as they are held in a special coded form within the program.

***REPLACE name1 name2**

Replace occurrences of 'name1' by 'name2'.

This routine searches the BASIC program currently in store and replaces all occurrences of the variable 'name1' by 'name2' - both parameters must be present. The restrictions on 'name1' are similar to those on the parameter used by *FIND except that keywords cannot be replaced. Again, only true variables or constants will be replaced as this example shows:

```
10 C = 1
20 C% = RND(6)
30 PRINT "Count = ";C
40 ACC = 66:REM SET ACC
50 IF C% < 3 GOTO 1000
60 C = C + 1
70 GOTO 30
1000 ACC = ACC/11:END
```

*REPLACE C Geoff will produce:

```
10 Geoff = 1
20 C% = RND(6)
30 PRINT "Count = ";Geoff
40 ACC = 66:REM SET ACC
50 IF C% < 3 GOTO 1000
60 Geoff = Geoff + 1
70 GOTO 30
1000 ACC = ACC/11:END
```

as it should.

Furthermore,

***REPLACE 1 one**

will only affect lines 10 and 60 - the '11' in line 1000 will be ignored (it is not a '1' so it should be ignored) and the assignment:

5 one = 1

will make the program function as before.

If replacing a short variable name by a longer one would cause any line to exceed 255 characters in length, further replacement is abandoned for that line and a 'Long at' message is issued. In this case, replacement carries on from the next line.

***XREF (N)**

Cross-reference current BASIC program.

This command produces an alphabetical list of all variables used in a BASIC program together with the line numbers at which the variables are referenced. It is not necessary to RUN the program before typing XREF as the variables are read from the code, not from BASIC's tables. To ensure a tidy display, only 8 characters are given for each variable name - if the name is longer, the remaining characters are replaced by a '*' symbol. However, XREF still holds the full name in its work areas and all references to that variable will be correctly identified.

The routine stores its internal tables at the current value of TOP, which is usually at the end of your program. This is where BASIC stores its variables so that, if it is possible to RUN a program, it will be possible to XREF it as the XREF tables will always be shorter than those compiled by BASIC. Typical output from the routine looks like this:

Alpha\$	130	4600	4610	4620	4700
	4890	5200	5750		
Count%	10	350	1000		
envelop*	80	2300	5670		
envelop*	300	320	340	360	
generat*	4005	4010	4020	4040	4060
x\$	30				

Here, envelop* has been substituted for the variable names 'envelope1' and 'envelope2', whereas generat* has replaced 'generation'. Notice how the two envelop*'s are still individually referenced.

If the 'N' parameter is used, only numeric items will be referenced - this will include both decimal and hexadecimal (preceded by '&') items in strict ASCII order. 'N' is the only valid parameter for this command - any other character will generate a 'Syntax' message.

No line numbers will be displayed in the column reserved for variable names; this is necessary because very long XREF's would otherwise be difficult to read.

If a printer is enabled via CTRL—B or VDU 2, then the routine will detect this and display the listing in an 80 column format so that 11 line number references will fit onto one line instead of the usual 5.

***TRON (lno)**

Enable enhanced BASIC trace.

BASIC provides a 'trace' facility which displays the number of each line as it is about to be executed. One of the practical problems with using this utility is that the output line numbers are mixed with any output from the actual program, making it very difficult to see what the program is doing. The **SLAVE** enhanced TRACE will confine the line numbers to the top left hand corner of the current text window, making the display much more readable. As with BASIC's trace, the 'lno' option is provided so that, if required, only line numbers less than it are displayed. The purpose of this feature is to enable you to trace the main section of your program without following each individual procedure or subroutine - these are usually located at the end of the program.

When *TRON has been enabled, each line number is displayed at the top-left of the screen before it is executed. The program is then halted. Pressing, and then releasing the CTRL key will cause that line to be executed, and a new line number to be displayed. Thus CTRL can be used to slowly single step through your program.

Holding down the SHIFT key will cause the program to run at slightly less than normal speed, displaying each line visited as it does so. Releasing SHIFT will halt the program once more. To leave the program you must type CTRL—ESCAPE.

*TRON shares the same code as BASIC's trace and it is subject to the same limitations. It works by detecting the line number delimiter characters '[' and ']' and an attempt to PRINT these (or VDU 91 or VDU 93) will trick the routine and may upset the display. This is not serious, but if it affects the tracing of your program, those lines should temporarily be removed. Similarly, spreading certain VDU statements (eg VDU 23) over more than one line will work provided nothing is output to the screen. In this case, it is almost certain that BASIC's trace will appear to hang up (which means that *TRON will fail too) - however, for most normal types of programs, the routine is easy and reliable to use.

Once you have finished with the **SLAVE** trace facility, you should issue the command *TROFF to cancel it.

***TROFF**

Disable enhanced trace feature.

This command will turn off both the enhanced and the normal trace. You should always issue this command after using *TRON even though the BASIC instruction TRACE OFF will apparently call off the trace. This is because *TRON sets certain 'vectors' that need to be reset before the computer can function properly. Alternatively, pressing 'BREAK' will do the job.

***CHAR (first) (last)**

Display character definitions.

This command has two functions:

i) *CHAR alone will display the full set of printable characters, that is those with ASCII codes in the range 32 — 255. The characters are printed in rows of 16 or 32 (depending on the screen mode) and the rows are separated by a single blank line.

Notice that CHR\$127 (the 'delete' character) is printed as a space, and that in MODE 7 the presence of teletext control codes will affect the display so that the character set shown is not the same as that given in the appendix of the User Guide.

ii) *CHAR followed by a decimal number in the range 32 — 255 will give the VDU 23 definition of that character in a form suitable for editing in command mode. VDU is abbreviated to V. (which is acceptable to BASIC) and, except for the 23, all other numbers are in hex, preceded by a '&'.

If a second parameter is given, the definitions of all the characters between (and including) the parameters are given. For example

*CHAR 65 66 will produce:

```
V.23,&41,&3C,&66,&66,&7E,&66,&66,&66,&00
```

```
V.23,&42,&7C,&66,&66,&7C,&66,&66,&7C,&00
```

which are the definitions for the letters 'A' and 'B'. These definitions can now be edited or copied as required.

***EDIT (Ino)**

Enter dynamic BASIC editor.

This environment is similar to a text editor where movement around a piece of text is controlled by the cursor keys; the difference here is that the 'text' is actually a BASIC program in store. The program may be listed either backwards or forwards and the line in the centre of the screen may be edited by typing in BASIC program text.

*EDIT displays the program stored at PAGE on a MODE 7 screen starting from the first line of the program. The optional 'Ino' parameter can be used to enter the program at any required line number.

Initially, the familiar underline cursor will be seen on the line at the centre of the screen - this means that the editor is in 'INSERT' mode and any text entered will shift the rest of the line to the right and insert text at the cursor position. Only the first 40 characters of each line are displayed, and any longer lines will appear to be truncated. However, as the cursor is shifted along the line, the rest of the line is brought into view.

Pressing 'TAB' will cause the cursor to be replaced by a flashing block - the editor is now in 'OVERSTRIKE' mode and any text entered will overwrite the text already present on the line. Notice that you cannot overwrite beyond the end of a line because there is nothing there to be overwritten!

Once you have entered the editor, the following keys are effective:

TAB	Toggles between INSERT and OVERSTRIKE modes
CURSOR UP	Scrolls the BASIC program downwards (so that the edit line effectively rolls upwards). Movement stops when the start of the program is reached.
CURSOR DOWN	Scrolls the BASIC program upwards (in this case the edit line appears to move downwards, in the direction of the arrow key). Movement stops when the end of the program is reached.
SHIFT	Holding down the SHIFT key in conjunction with cursor up or down will cause the movement to take place twelve lines (half a screen) at a time.

CURSOR LEFT

Moves the edit cursor one character to the left - this may cause the line to scroll in its window. The movement is arranged so that characters to the left and right of the cursor may be clearly seen. Movement stops when the start of the line is reached. Note that it is not possible to edit the line number as that area is protected by the editor.

CURSOR RIGHT

Moves the cursor to the right by one character position. As the end of the line approaches, this may cause the whole line to scroll to the left.

RETURN

Passes the edit line (the one containing the cursor) to BASIC so that it is entered into the program. This is the ONLY way to enter a line - leaving a line by scrolling will cause the original line to be retained.

DELETE

Operates in the usual way, that is, it causes the character to the left of the cursor to be deleted and the remainder of the line is moved back by one position.

SHIFT

When used in conjunction with the DELETE key, this enables the editor to perform forward deletion. This means that the character AT the cursor position is deleted before the rest of the line is moved back.

COPY

If you wish to recall a line that you have amended (perhaps in error) then pressing the COPY key will return the original line to you and place the cursor at the start of it ready for re-editing.

CTRL-A

Adds a line to the program following the edit line. The number of the line so inserted is one greater than that of the current edit line. To insert a block of code into a program the best method is to generate a number of blank lines first, using the CTRL-A facility and then to scroll back to type in the text. If this feature causes a line to duplicate an existing line number then the original line is deleted from the program (as it is when AUTO is used). The lines generated by this facility initially consist of a line number followed by a space.

CTRL-K

Deletes the edit line from the program. Once deleted, the line cannot be retrieved.

ESCAPE

Leaves the editor and returns to BASIC.

The facilities offered by the editor make the editing and listing of BASIC programs very straightforward and, once mastered, they represent a great improvement on the traditional LIST and COPY method of editing that is standard on the BBC computer. Some practice is needed to understand the use of the various keys but this will be found to be time well spent considering the flexibility allowed by the editor.

Entering the editor has no effect on cursor repeat speed and you might like to set it higher than its usual value by issuing a *FX 12 3 command before typing *EDIT. This has turned out to be an ideal repeat speed for the routines.

Machine code and disc utilities

***MOVE first +len/last to**

Move a block of memory.

Each parameter is given in hex (without the leading '&') and is used to define the block of memory to be moved and the destination address. The syntax of the command has been chosen to be consistent with the Operating System's ***SAVE** command in that either the length of the block (by preceding it with '+') or the last byte to be moved can be specified.

Unless you are familiar with the memory of the computer you should be careful where you move bytes to - it is safest to restrict the parameter 'to' to lie between E00 and 7FFF. As with all other **SLAVE** commands that refer to readable memory, the currently selected (via ***CROM**) ROM will be used if the range of the move extends beyond &8000.

For Example:

```
*CROM BASIC
*MOVE 8000 +4000 2000
```

will transfer the contents of the BASIC ROM into RAM starting at location &2000.

Here are some valid ***MOVE** commands:

```
*MOVE B00 +FF 1900    - move 255 bytes
*MOVE 4000 +2000 6000  - copy top half of MODE 3 screen to bottom half
*MOVE 1900 +0 1300     - move NOTHING
*MOVE 7C00 7FFE 7C01   - shift MODE 7 screen one character right
*MOVE A% +45 B%
*MOVE X% Y% Z%
*MOVE 900 +1 90A       - move one byte
*MOVE 900 900 90A      - IDENTICAL to above command
```

The fourth example shows that the move is 'intelligent' and can cope with overlapping areas.

Examples 5 and 6 illustrate the use resident integer variables as parameters to **SLAVE** commands. ***MOVE** can be used in this way to move a 'sprite' around the screen much quicker than it can be done in BASIC — this is useful for video games.

A limit of 32k (+ 7FFF) bytes has been placed on the size of the block that can be moved and any attempt to move more than this amount will generate a 'Too many bytes' error message.

***CROM (no/rsp)**

Choose, or inspect, sideways ROM

There are two forms of the *CROM command:

i) *CROM without a parameter will display the contents of the computer's 'sideways' ROM sockets.

ii) *CROM followed by a decimal number in the range 0 - 15 OR by a ROM specification will 'page-in' that ROM, AS FAR AS SLAVE IS CONCERNED.

If your computer is fitted with a 'sideways' extension board then *CROM will produce something like this:

```
0—DFS,NET
* 1—BASIC
2—SLAVE
3
4
5
6 FORTH
7
8
9
10
11
12
13
14—FORTH
15
```

The computer recognises 16 sockets, only one of which can be 'paged-in' at any time - under normal circumstances this will be BASIC. The 16k of memory occupied by each sideways ROM is always located at addresses &8000 to &BFFF and inspecting this area of memory using **SLAVE** will display the contents of the chip indicated by the '*'. In the example above, this will be BASIC, (which is the default setting by **SLAVE** when the computer is first switched on, or following a 'BREAK'.

If your computer has no sideways extension fitted then the contents of the sockets will be repeated, like this:

```
0 BASIC
1 MYDFS
2 SLAVE
```

```

3 DFS,NET
4 BASIC
5 MYDFS
6 SLAVE
7 DFS,NET
8 BASIC
9 MYDFS
10 SLAVE
11 DFS,NET
*12—BASIC
13 MYDFS
14—SLAVE
15—DFS,NET

```

Those chips marked with a '—' are recognised by the Machine Operating System and may be paged in by it to process '*' commands, interrupts and various other system calls.

In the example above, the computer is fitted with two DFS chips, only one of which is recognised by the computer. Issuing any '*' command recognised only by the MYDFS chip would fail as the computer is not aware of the chip's presence. However, **SLAVE** can still read the chip in the same way as all the others.

The second form of the *CROM command enables **SLAVE** to inspect any chip in the sideways sockets. Taking the first example above, where **SLAVE** is in socket 2, the following instructions will all have the same effect:

```

*crom SLAVE
*CROM SL.
*CROM S
*CROM 2      or even:
*CROM R%     where R% has previously been set to 2.

```

This is a useful feature if you do not know which socket a particular chip occupies - for example, if a program that contained **SLAVE** commands was moved onto a different machine also fitted with **SLAVE**. The effect of this second form of the *CROM command will be to move the '*' to the new chip you have 'paged in'. This ROM will remain paged in until you press 'BREAK' or turn the power off - IT IS PRESERVED EVEN ON EXIT FROM SLAVE.

The *CROM command is very powerful when used in conjunction with some of the advanced **SLAVE** facilities mentioned later in this section, and this point should always be borne in mind when using **SLAVE** commands:

The chip indicated by the '*' on the crom screen will always appear to be present when executing **SLAVE** commands that refer (even indirectly) to memory. Thus-

MOVE, GET, PHEX, MON, HEX, DIS, DEBUG, CRC and REL will always consult the chip in the '*' socket whenever memory references in the range &8000 - &BFFF occur, and this process is totally transparent to the user. This feature, unique to **SLAVE** is both flexible and yet simple to control. Try this example:

```
MODE 7
*CROM BAS.
*MOVE 8000 + 300 7C00
```

This will fill the screen with what appear to be random characters, but on closer inspection, you will see some of the familiar BASIC keywords. The block of memory moved to the screen is the first three pages (= 3 x 256 bytes) of the BASIC ROM. If you now type:

```
MODE 7
*CROM S.
*MOVE 8000 + 300 7C00
```

you will move the first three pages of **SLAVE** onto the screen (you should be able to make out the word '**SLAVE**') even though you have (apparently) moved the same block of code.

***EDKEY (first) (last)**

Display function key definitions

The contents of the selected function keys are displayed in a suitable format for screen editing using the COPY key. Also displayed is the number of bytes in the function key buffer area free for future definitions. If no parameter is used, the contents of all 16 keys are listed. Otherwise, 'first' is a decimal number in the range 0 - 15 indicating the first key to be listed - if no other parameter is passed, only that key is listed. The optional second parameter, 'last' (also 0 - 15) is the last key to be listed.

The routine has been carefully designed to deal with ALL definitions, including some of the more awkward ones, for example:

```
*KEY 2 "|||?"  
*KEY10 "|||||"
```

This last definition is perfectly correct syntax, although most function key editors will list it as ' *KEY10 "|||||"' which will generate a 'Bad string' message if you attempt to COPY it from the screen.

***SLOW (dly)**

Slow processor.

This command causes the entire computer to slow down. If no parameter is passed, or if *SLOW 0 is used, the computer will revert to normal speed. 'dly' is a decimal number in the range 0 - 255 which determines the degree of slowing down - 199 generally giving the longest delay. Values of 'dly' above 199 can slow the computer down so much that it stops altogether! The scale is not linear and the actual effect depends on what the computer is being asked to do but nonetheless, this facility is a powerful debugging aid.

*SLOW uses the same mechanism to operate as does the system clock and keyboard auto-repeat, and will appear to speed-up TIME as measured by the internal clock. Following a *SLOW command the keyboard auto-repeat is disabled with a *FX 11 0 Operating System call and the keyboard will not behave in quite the usual way, although it has been contrived to function as well as possible under the circumstances.

A simple example of the command in use is:

```
MODE 5
*HELP MC.
*SLOW 197
*MODE 2
*SLOW
```

When you have entered the 'MODE 2' command you will see a slow motion clear screen (and mode-change) effect - notice how the screen is cleared in blocks rather than in one flowing movement. Also, if you watch the '>' prompt as it is printed at the top of the screen you will see that it is printed from the bottom upwards - this is impossible to see unless you can slow the processor right down, as with this command.

*SLOW is useful in debugging graphics programs as it can reveal the order in which things are printed on the screen as this simple example shows:

```
*SLOW 198
*DEBUG
```

Low values of 'dly' can also be used to slow down video games to the point where they are playable!

***GET string/'hex'**

Find string or bytes in memory.

This command will search memory for all occurrences of a character string and/or sequence of bytes. 'string' can contain any characters that can be typed in at the keyboard except for ' and space. Character codes outside the ASCII range can be searched for by entering the code in hex, surrounded by single quote marks. Each hex byte must comprise two hex digits using characters 0 - 9 and A - F (upper case).

When *GET is called from BASIC (or command mode in response to a '>' prompt), **SLAVE** will print the address, in hex, at which each occurrence of the string is found. (That is, **SLAVE** will indicate where the first byte of the matched string is, when the string is found in memory).

The hash character '#' can be used as a wildcard and will match any character in that position of the string. Both character and hex items can be freely mixed in any search as will be seen in the examples below.

When called from the **SLAVE** HEX or MON routines, GET performs a slightly different function which is explained in the description of those routines.

Here are some valid *GET commands:-

*GET BASIC

*GET B'41'SIC

*GET '4241'SI'43'

*GET B##'4943

- each of these commands will find all references to the word 'BASIC', while the last one may find other strings as well.

These commands are illegal:-

*GET '

*GET AB CDE

*GET A'123'

*GET '0A'123'Y4'

*GET '31##212'PK

The area below address &800 contains various buffers used by both **SLAVE** and the Operating System and any strings located below this address should be ignored.

***PHEX first +len/last**

Dump memory to printer

The block of memory specified by the parameters is dumped to an 80-column printer in standard hex dump format i.e. each line having an address, 16 bytes in hex and, where appropriate, their ASCII representations. Issuing the command will turn on the printer, and dump bytes from memory location 'first'; when the dump is complete, the printer is turned off. During the dump, nothing is output to the screen.

The first line of print will contain 'first' which can be valid address (in hex) and the last line to be printed will contain location 'last'. Alternatively, if you know the size of the area you want to list, use the '+ len' parameter by specifying the number of bytes to dump (also in hex).

To halt the print at any time, press the ESCAPE key. This will cause the routine to quit once the current line has been completed.

***GO addr**

Execute machine code.

This command is identical to the BASIC instruction 'CALL', which causes the computer to branch to a piece of machine code and return to the code following 'CALL' on receipt of a 6502 'RTS' instruction. 'addr' is the address - in hex - of the code that is to be run.

*GO is needed when BASIC is not available, for example, when the computer is fitted with some other language chip, or more probably, when it is required to execute a piece of code from within **SLAVE's** 'No-Language' environment.

MEMORY EDITING

The next two utilities, HEX and MON, provide a powerful memory editing environment and this section describes the facilities provided by the editor.

The input line at the bottom of the screen shows the address at which the entered data will be located; this address will be highlighted in the main display. Data types can be freely mixed in the input line and are generally recognised by their lengths - each item should be separated from the next by one or more spaces. The editor will accept the following input types:

i) An address. Addresses should be FOUR characters long and entered in hex (ie. they are composed of any four characters from the ranges 0 - 9 and A - F). Entering an address on its own will reset the current location to that point. If the address is followed by data, then the data will be stored there and the location pointer updated accordingly.

ii) Hex data. Hexadecimal data bytes are recognised as being TWO characters in length.

iii) ASCII. Any character that can be typed at the keyboard can be stored at the current location. Such items are just ONE character long. If you wish to store a character string, rather than enter it in single characters separated by spaces, you may type a quote mark ' ' and follow it with the string. This string may include spaces or further quote marks - any text that is entered will be stored exactly as it has been typed.

iv) Assembler mnemonics. **SLAVE** contains an assembler and will accept any of the 6502 mnemonics, followed by an operand, where appropriate. Assembler items will be THREE characters long although the operand need not be. All numeric operands are taken to be hexadecimal, but the leading '&' character is optional. For branch instructions, the operand will be the destination address (in hex) - **SLAVE** will compute the correct displacement.

v) Command lines. Any Operating System command (commands preceded by '*') can be entered at the edit line and will be passed by **SLAVE** to the Operating System. If the command produces any output, the screen will clear to allow for it and the message '<RETURN>' will appear at the bottom of the screen. Pressing the RETURN or ESCAPE keys will return to the previous editing prompt. Anything following the '*' is taken by the Operating System to be part of the line, so in this case, it is not possible to enter other data types after a '*' command, without first pressing RETURN. All **SLAVE** '*' commands can be entered without confusing the system - for example a '*MON' call from the HEX routine will enter the mnemonitor

(MON routine) at the correct address. Subsequently pressing ESCAPE while in the mnemonitor will return you not to BASIC, but back to the HEX routine you originally came from. Whereas this sequence of events is valid, the next input type renders it unnecessary.

vi) CTRL - D, H, M. To change from any of the routines: DEBUG, HEX or MON to any of the others, simply type CTRL - (initial letter of routine you wish to enter). Nothing is affected in that the editing address stays the same and any DEBUG parameters will be unchanged. This is very useful for DEBUG as it does not have quite the same editing facilities as HEX and MON; in particular, it cannot be used to assemble code.

vii) CTRL—U and DELETE. These function in the normal way and are used to delete the entire input line, and the last character entered respectively.

viii) RETURN. Causes the input line to be processed in the usual way.

ix) ESCAPE. Pressing the ESCAPE key will return you to the routine from which the current processing routine was called. As we have seen, this may or may not be BASIC.

If **SLAVE** encounters any invalid input items, the beeper will sound and that item, together with any that follow it on that line will be ignored. Once editing is complete, the bottom line will clear and the main display will be updated with the new editing address being immediately after the last byte changed.

Here are some examples of valid edit lines:

8528:0980 00 50 E r r o r 00 8528 or, alternatively:

8528:0980 00 50 "Error

0987:00 8528

The next example will insert a subroutine at location &980 (you may call this from the mnemonitor with '*GO 980' - to escape from the routine press TAB — DO NOT press ESCAPE.

0980:JSR FFE0 CMP #9 BNE 098F

0987:LDA #&7 JSR FFEE JMP FFE7

098F:JSR &FFEE JMP 980

Use of '*' commands:

0900:LDA #7 00 *MON 900

Mixing input types:

1900:TAX 12 23 a b c d ASL A 1330 +

***HEX (addr)**

Enter hex display editor.

With no parameter, *HEX will enter the hex dump at the previously-used address. If this is the first entry to a **SLAVE** machine code routine, this address will default to &8000. The optional parameter 'addr' can be any valid address and must be entered in hex (but without the leading '&').

The MODE 7 display consists of three sections:

i) A status line showing the current address, together with the byte at this address in hex, decimal, as an ASCII character and either the binary, BASIC token, or assembler mnemonic (with operand if appropriate) represented by that byte.

ii) The hexadecimal contents of the memory around the current address, displayed 8 bytes to a line. Printing characters (those with ASCII codes in the range 32 - 126) are shown in cyan, while non-printing characters are shown in magenta. The byte currently being edited is shown in yellow.

iii) The corresponding ASCII representations of all printable characters are shown on the right of the screen.

All of the comprehensive editing facilities described in the previous section are available. Other keys operate as follows:

CURSOR LEFT Move editing (highlighted) cursor one byte to the left.

CURSOR RIGHT Move editing cursor one byte to the right.

CURSOR UP Scroll display down by one line so that the editing location is now 8 bytes below its previous value.

CURSOR DOWN Scroll display up by one line so that the editing location is now 8 bytes above its previous value.

Auto-repeat is unaffected by **SLAVE** and will still operate at the speed it was last set to by any previous *FX 12 command. This will determine the speed of the scrolling.

CTRL When used in conjunction with the up or down arrow key, this causes the scrolling to take place very rapidly - half a page (128 bytes) in the relevant direction.

TAB

Continually refreshes the screen and so reveals any dynamic (changing) locations. Normally, the screen is plotted once on entry to the hex dump and never changes until you cause the routine to re-read memory. In fact, many locations within the computer are constantly being updated and holding down the TAB key while the display is stationary will reveal these locations. To see this facility in use, try *HEX 2A0 and then hold down TAB.

SHIFT

This key acts as an 'accelerator' and will cause any of the above operations to take place at a much higher speed.

CTRL-TAB

Toggles the 'Binary' field in the status line between 'Binary', 'Token' and 'Mnemonic'. In 'Token' mode, this field will display the BASIC keyword associated with the hex byte at the editing cursor (if there is one - otherwise the field will be blank). A list of these tokens is given in the User Guide appendix and this facility will be found to be extremely useful for studying BASIC programs as all the keywords in them are held in this tokenised format. **SLAVE** will decode them for you. In 'Mnemonic' mode, the byte at the editing cursor is treated as a 6502 opcode and is disassembled accordingly.

COPY

This facility is only available in conjunction with *GET. To locate strings in memory the *GET command can be called from within the HEX editor. When called in this way, the search stops when the string is found and **SLAVE** will re-enter the editor with the editing cursor placed on the first byte of the string. Pressing COPY allows the search to continue from this point so that the next occurrence may be highlighted. COPY will still function even if you scroll the display with the arrow keys - in this case, the search is resumed from the editing cursor, not from the last occurrence of the string.

ESCAPE

Quit the HEX routine and return to the calling environment (not necessarily BASIC).

***MON (addr)**

Enter the mnemonitor.

The mnemonitor (MNEmonic MONITOR) serves the same function as the hex dump routine just described except that the main display is composed of disassembled 6502 opcodes rather than hex.

*MON includes the usual memory editing features as well as these control keys:

CURSOR LEFT Decrement the current address by 1 and replot the screen.

CURSOR RIGHT Increment the current address by 1 and replot the screen.

SHIFT Used in conjunction with either of the above keys causes the 'stack' to be activated. If the operand for the edit (highlighted) instruction is preceded by an '&' character - in other words, it references a specific memory location; SHIFT — CURSOR RIGHT will 'follow' that address and save the current address on its stack. The stack is displayed in the top right hand corner of the screen and shows the 'return' address and the depth of the stack (number of items below the top one, which is displayed). No more than eight such addresses can be stored. If SHIFT — CURSOR RIGHT is pressed with the counter on 7 (one item on top with seven below it, total eight), the 'follow' will occur but the stack depth and last address will remain the same.

SHIFT — CURSOR LEFT will unstack an address and return you to the point you came from originally. This facility is therefore extremely useful for following a long piece of code full of branches and JSR's as it is impossible to get lost provided you use the stack to trace the flow of the code. Whenever the stack is used the entire screen display is updated.

CURSOR UP Scroll the screen down by one line (equivalent to one 6502 instruction) this brings the PREVIOUS instruction onto the editing line and fetches a new instruction in at the top of the screen. In other words,

SLAVE can disassemble backwards! This advanced (and very rare) feature makes the disassembly of a piece of code as simple as following the usual style of scrolling hex dump.

CURSOR DOWN

Scroll the screen up by one line (6502 instruction) and fetch a new instruction in at the bottom of the screen. Most disassemblers can scroll forwards in this way.

SHIFT

With either of the two scrolling keys, SHIFT acts as an 'accelerator' to enable very fast scrolling.

COPY

As with *HEX, it is possible to call the **SLAVE** 'GET' routine from the mnemonitor. Once the string is found, it is identified within MON and COPY can be used to fetch the next occurrence. Pressing COPY will continue the search from the cursor position.

ESCAPE

Quit the mnemonitor.

DISC ROUTINES

The next three utilities are only of use to disc users. If your computer is not fitted with a disc interface, or you are currently using the cassette filing system, a call to any of these routines will result in a 'No DFS!' message.

***SPACE (drv)**

Report free disc space.

This command displays - in hex - the number of free sectors on a disc, together with the largest 'extent', that is, group of consecutive sectors. The first number is the amount of disc space that will be available after *COMPACTing the disc, while the second is the size (in pages) of the largest file that can be saved.

If no parameter is specified, then the currently selected drive is used. Otherwise, 'drv' can be a drive number in the range 0 - 3.

Example: *Space 2

may give: Sectors Free: 05B

 Max Extent: 027

This shows that drive 2 has 91 free sectors (after compacting, these sectors will be together at the end of the disc) but the largest free area is only 39 sectors long. If you wish to save a BASIC program onto this disc, check the value of ~TOP-PAGE to find its length. Suppose this was 254A; in this case, the program will just fit onto the disc as it requires &26 free sectors. On the other hand, if the value was something like 34C8, then you would get a 'Disc full' message, meaning that there is no suitably large extent to hold the program, although there is clearly enough space available.

When using this facility remember that the *COMPACT command may overwrite memory and so you should save valuable programs on a work disc before issuing it.

If the two numbers produced by the *SPACE command are the same, then there is little point in *COMPACTing it as no more space will be made available. However, the command may rearrange the files on the disc so that they will lie together at the start of the disc.

***DGET string/'hex'**

Find string or bytes on disc.

The disc in the currently selected drive is scanned from Sector 0, Track 0 for the sequence of bytes specified. When a match is first found, the routine enters the **SLAVE** disc sector editor, which is the subject of the next section. Before entering the editor, you may be asked if you are prepared to change to MODE 3, depending on the MODE you are in when you issue the *DGET command.

Once you have entered the editor, subsequent occurrences of the string can be located by pressing the 'COPY' key. The search resumes from the current cursor position, even if you have changed track or sector.

The scope of the search is limited to sectors and does not cross sector boundaries. Thus, if you search for the string 'COMPUTER' the routine will report a possible match if, say, the characters 'COMP' occur at the very end of a sector. To see if the match is genuine, you will have to read the next sector (using CTRL → from within DISCED) to see if the remaining characters match.

The parameter passed to DGET has EXACTLY the same format as that used with the *GET command (hex and character types can be mixed, wildcard facility, etc.) and some examples of valid syntax are given with that command.

***DISCED (drv)**

Enter disc sector editor.

The **SLAVE** disc sector editor allows any sector of a disc to be read, edited and re-written to disc. It will function with any single or double-sided, 40 or 80 track, single density disc in ACORN compatible format. If no drive is specified, the currently selected drive is used. Otherwise 'drv', a number in the range 0 - 3, can be used to select a drive.

A whole sector is displayed on a MODE 3 screen. If the computer was in a MODE requiring less than 16k of memory (ie. MODE 4, 5, 6 or 7) when the command was issued then the question: 'MODE 3 - OK?' will be asked. A response other than 'Y' will cause the editor to be abandoned, without affecting memory in any way. If you answer 'Y' then the editor will enter MODE 3, possibly overwriting any long BASIC program that is in store.

Because a MODE 3 screen can be difficult to read on a television set, or certain types of monitor, an extra feature has been included to allow you to select foreground and background colours. Entering the editor from MODE 3 will not cause a MODE change but will simply clear the screen. In this way you can select your own palette for the display by means of the VDU 19 command. For example, if you find blue on white easy to read, enter the disc editor like this:

```
MODE 3
```

```
VDU 19,0,7;0;19,1,4;0;
```

```
*DISCED
```

On entry, the editor will read the catalogue of the disc and display the contents of Track 0, Sector 0. The display consists of three sections:

- i) A status line at the top of the screen.
- ii) A hex display of the contents of the sector.
- iii) The ASCII representations (where appropriate) of the sectors contents.

Three cursors will be seen - one in each of the above areas.

The status line displays the following information:

```
Drive: 0  Track: 0  Sector: 0  Abs Sector: 000  Edit Mode: HEX
```

Pressing the TAB key will move the status (flashing) cursor between these fields (and SHIFT-TAB will move it in the opposite direction), and any field may be amended by entering a new value.


The 'RETURN' key has two functions: firstly, when the status cursor is in the 'Edit Mode' field, it acts as a toggle between hex' and 'ASCII' and secondly, when the status cursor is in any other field, it causes the newly selected sector to be read from disc. If the sector's contents have been altered in any way, you will be asked 'Update Disc?'. You should respond 'Y' if you wish to write the sector back to (where it came from on) the disc, or 'N' if you wish to read the new sector without replacing the current one. Any other response will cause the attempted move to be abandoned.

The status field at the flashing cursor can be amended by typing new values into it. These are vetted as they are entered so that they always lie within the boundaries of the disc; in some cases 'wraparound' may occur. 'Drive', 'Track' and 'Sector' are all decimal numbers with the ranges: 0 - 3, — 0 - 39 or 0 - 79, and 0 - 9 respectively. The 'Track' range is determined by the type of drive you have - either 40 or 80 tracks. It is possible to mix drives in that you can edit a 40-track disc in drive 0 and then read drive 1 which contains an 80-track disc - **SLAVE** will detect this and respond accordingly.

'Abs Sector' is given in hex (usually this field will be read from the catalogue in Track 0, Sector 1, where the DFS stores it in hex) and can be edited like all the other status fields. While it is being amended, Track and Sector are also updated to be consistent with it - a change to any of these fields will cause an immediate update of the other two.

For 40-track discs, the largest possible entry in this field is 18F while for 80-track discs it is 31F.

When the status cursor is in the 'Edit Mode' field, typing in data at the keyboard will cause the contents of the sector located at the block cursors to be amended. If the mode is 'HEX', any input will be regarded as a hex digit which will be shifted into the right of the byte under the hex cursor. Pressing the space bar will move the cursors to the next byte. If the mode is 'ASCII', typing any character except 'TAB' or 'RETURN' will cause that character to be substituted at the editing cursor position. In either case, both displays are updated together and show the sector as it will be written to disc if you choose to do so.

The 'ASCII' mode is very flexible and will allow you to enter 'control' characters as well as printable ASCII. For example, to fill an area of the sector with hex '00', hold down CTRL— while you are in ASCII mode and let the auto-repeat do the rest.

When the disc editor has been entered via the DGET routine, the 'COPY' key is used to locate the next occurrence of the search string beyond the current cursor position. Otherwise it has no effect.

Whichever mode you are in, the arrow keys, in conjunction with CTRL and SHIFT have the following functions:

Cursor up	— Move editing cursors up one line
Cursor down	— Move editing cursors down one line
Cursor left	— Move editing cursors one byte to the left
Cursor right	— Move editing cursors one byte to the right

SHIFT and:

Cursor up	— Move editing cursors to top line
Cursor down	— Move editing cursors to bottom line
Cursor left	— Move editing cursors to far left of current line
Cursor right	— Move editing cursors to far right of current line

CTRL and:

Cursor up	— Read this sector on previous TRACK
Cursor down	— Read this sector on next TRACK
Cursor left	— Read previous SECTOR (probably still on this track)
Cursor right	— Read next SECTOR (probably still on this track)

To leave the disc editor, press the 'ESCAPE' key. Please note that this will cause an immediate exit from the routine and you will lose any amendments you have made to the current sector. If you wish to keep the amendments, then you will have to write the sector before escaping - the easiest way is to use CTRL → to read the next sector. Respond to 'Update Disc?' with 'Y' before you quit the editor with 'ESCAPE'.

To use the editor effectively you should be aware of the format of the information that is stored on the disc and that is beyond the scope of this manual. Your disc manual should describe this format - especially the layout of the two catalogue sectors, which are probably the two most important sectors on any disc. If the catalogue is severely corrupted, or is not in the standard ACORN format, then **SLAVE** will probably fail to read the disc and will generate a 'Disc Error' message.

***DIS (first) (+len/last) (sw)**

Disassemble memory to current output channel(s)

*DIS will produce a disassembly listing of any part of RAM or ROM (as selected by *CROM) in any chosen format. The block of memory is defined by the parameters 'first', '+ len' and 'last' - see the description of *MOVE to see how these are used.

If no parameters are given, then 'first' will default to the last address read by one of **SLAVE's** machine code routines (eg. HEX) and disassembly will continue until 'ESCAPE' is pressed.

The third parameter is a switch byte that determines the fields to be printed and must be given in hex (without the leading '&'). Thus it lies in the range 0 -FF. If 'sw' is thought of as an 8-bit binary number each bit corresponds to a field that may or may not be output by the disassembler. The settings of the bits are as follows:

FIELD	BIT	HEX VALUE
Hexadecimal address	7	80
Hex contents of address	6	40
Sequence numbers	5	20
Labels	4	10
Opcodes (and operands)	3	08
ASCII	2	04
Strip spaces between fields	1	02
Unused	0	01

If 'sw' is omitted it will take the default value of &CC which will give address, hex, opcodes and ASCII. The fields are printed from left to right in the order defined by this table. This makes it easy to select the right bit-pattern for 'sw' by matching each bit with its associated field. For example, when code is being SPOOLED out to disc or tape, the fields required will be: sequence numbers, labels, opcodes and space-strip and a setting of (hex) 20 + 10 + 08 + 02 = 3A will achieve this combination.

Space-strip is useful for reducing the size of very long listings. Normally, the various fields will be separated so that the output lies in neat columns - this is very wasteful of space in files and can greatly increase the time required to print a disassembled listing. Setting the 'Space-strip' bit in 'sw' will omit the padding spaces and result in much shorter (though slightly less readable) files.

When bit 4 of the switch is set, a three-pass labelled disassembly will take place. During the first two passes no output is produced but the number of bytes in the symbol table (pass 1) and the number of labels finally used (pass 2) will be displayed on the screen. The disassembly output is produced during pass 3.

Each label is composed of 5 characters: a single alphabetic identifier indicating the label type, and four hexadecimal digits forming the address of the label. Thus, labels are unique, whatever area of memory is disassembled. These label types are supported ('h' stands for any hex digit):

Ohhhh	Disassembly origin
Dhhhh	Data (subject of Load or Store instruction)
Jhhhh	JMP or JSR destination
Bhhhh	Branch destination
Thhhh	Table referenced via indexed addressing
Vhhhh	Vector (indirect jump)

Wherever an absolute operand may be replaced by a label in the opcode column, the substitution will take place so that the only memory referenced by the disassembled code will lie outside the boundaries of that code. This means that the code should reassemble correctly when required.

If reference is made to a byte in the middle of an instruction (for example the high byte of a LDA absolute instruction) then the instruction will be labelled and the reference will appear as 'label + 2'.

Label definitions will be preceded by a '.' character (as used in BASICs assembler) and followed by a single space. Unlabelled opcodes will be preceded by 7 spaces, unless the space-strip bit is set.

The work area used by *DIS is located at the current value of LOMEM, which usually points to the end of the current BASIC program. This is generally a safe area to store data, as BASIC will use it to store variables when the program is running. LOMEM has been chosen because it can be set in command mode (eg. 'LOMEM = &5000) or from within a BASIC program to point to any area of RAM - even that reserved for the screen. This work area is used by the routine to build its label tables and so LOMEM should be chosen carefully if you intend to produce a very large labelled disassembly. The size of the area obviously depends on the number of labels used but as an example, a disassembly of 16k of object code required 3.5k for the tables (and the first two passes took around 50 secs for table compilation).

Example of DIS

DIS may be used to SPOOL out files that can later be amended and re-assembled. This feature is probably more useful to you if you have a disc system, but it will work equally well with tapes, only rather slower. The example below shows how to create a BASIC program that consists of re-assembled code taken from **SLAVE's** 'output the contents of the accumulator in hex' routine.

*CROM SLAVE	Enable SLAVE to read itself
*SPOOL 'Outhex'	Open output file
*DIS 803F + 15 38	List code to screen and to file
*SPOOL	Close the output file

(At this point, the code is stored in the file 'Outhex' in ASCII format)

*EXEC 'Outhex'	Read the file back as though you had just typed it in
----------------	-------------------------------------------------------

RENUMBER Number it in 10's rather than 1's

1 DIM Code 50	Surround the source code with a
2 FOR pass=0 TO 3 STEP 3	BASIC program that can reassemble it
3 P% = Code	
4 [OPT pass	

500]

600 NEXT

RUN Regenerate the code in a new location

A% = &BA : CALL Code Try it out!!

This sequence of instructions should result in 'BA' being output from the new piece of code stored in RAM. Of course, once the code is in the form of a BASIC program, it may be edited and saved as required.

SPOOLING a very long piece of code could result in '???' opcodes (invalid 6502 instructions) appearing as **SLAVE** attempts to make sense of tables or areas of data. If your computer is fitted with BASIC 2, then you can type:

*REPLACE ??? EQUB

so that the code will re-assemble on RUN. This works because **SLAVE** prints the hex equivalent of the invalid byte following the '???' and 'EQUB' is an assembler directive that will insert hex bytes into the code. If you do not have BASIC 2, you will need to change any invalid opcodes by hand.

***DEBUG (addr)**

Enter machine code single step/debug routine.

This is a complex utility that allows you to execute machine code subroutines and programs under controlled conditions. While the program is executing, registers can be altered, instructions skipped, memory examined etc. Some of the facilities offered by DEBUG are very advanced and you should study this section and the example at the end carefully before trying anything too adventurous.

The display is made up of four separate sections:

- i) Hex and character representations of six separate sections of memory together with the stack.
- ii) Processor and DEBUG status information.
- iii) Disassemble code around the current instruction.
- iv) An input line for adjusting DEBUG's parameters (this line looks like the HEX and MON input lines but it responds to different commands).

The program counter is not shown, but is the address at the start of the input line and also the central address of the disassembled display.

'addr' is an optional parameter and specifies the hex address at which DEBUG will commence - this will be placed in the program counter on entry to the routine. If 'addr' is omitted, it will default to the last address read by **SLAVE** in any of its machine code routines.

Once DEBUG has been entered, it will respond to a number of control keys and commands. We will look at the valid control keys first:

CURSOR UP — Scroll disassembled listing backwards (and update program counter accordingly). This is exactly the same effect as in MON.

CURSOR DOWN — Scroll disassembly forwards, and update program counter.

SHIFT — Pressing SHIFT at the same time as one of the two scrolling keys will speed up the scrolling display.

RETURN — Execute machine code under control of DEBUG until either BRK, a breakpoint or an invalid opcode is reached or until ESCAPE is pressed.

SHIFT/RETURN — Execute ONE machine code instruction. This is used to single step through a piece of code.

ESCAPE — While a piece of machine code is running, ESCAPE will halt execution and return control to you so that you can adjust parameters or study the results to date - 'SHIFT' will also have this effect.
'RETURN' will allow the code to continue running from the next instruction. If the code stops because it hits a certain type of instruction, or a breakpoint, the beeper will sound and ESCAPE will restore control to you. Pressing ESCAPE at other times will quit the DEBUG routine and return you to the calling environment.

CTRL-M or H — These have their usual meanings and will enter MON and HEX respectively.

Data entered on the input line are used to adjust the DEBUG parameters or the processor status. The basic format of a command is:

(initial letter) (new value)

No spaces should occur between the two fields. Several commands can be input on one line and will be processed from left to right. We will look at these commands in turn, moving down the screen.

Mn operand 'n' is a decimal number in the range 0 - 5 and 'operand' must conform to one of the valid 6502 addressing modes. The operand will be placed in the correct 'Ef.Adr' (Effective Address) slot at the top left of the screen (if n = 2, the third line will be used, and so on). The remainder of the line will be determined by the effective address.

The next column shows the address that is referenced; this is followed by the contents of the two preceding locations, the byte AT that location (highlighted) and the contents of the next 5 locations. This display is given in both hex and ASCII (where appropriate). The bottom line of the hex display is dedicated to the stack and the highlighted byte is that indicated by the stack pointer (ie. where the next byte pushed onto the stack will go - the last item pushed on the stack is the byte following this one in the display).

Here are some valid examples of this instruction:

M1 (20E)
M2 8000
M3 &700,X
M4 (70),Y
M5 (80,X)
M0 &12 etc.

- Ah** 'h' is a single hex byte. The accumulator is set to the value 'h'. The left hand side of the status display shows the accumulator and X and Y registers in both hex and decimal (underneath). When code is executed A, X and Y will be set to their displayed values.
- Xh** Set X-register
- Yh** Set Y-register.
- Sh** Set processor status register. The condition of each of the 6502's flags is shown beneath the values of A, X and Y using the notation: N=negative flag, V=overflow flag, *=unused, B=break flag, D=decimal mode, I=interrupt disable flag, Z=zero flag and C=carry flag. Where a particular flag is not set, the character is dropped from the display.

For example, S83 will set the N, Z and C flags (and unset the rest).

- Lnn** Set 'Levels' to 'nn' must be a decimal number in the range 1 - 99. Whenever a subroutine is called via a JSR instruction, 'Levels' is decremented by one to show the 'depth' the code has reached (see the explanation of the 'stack' in the section on *MON. If levels ever reaches 0, further subroutine calls are still executed but are not single stepped.
- Fhhh** Set 'Fence' to hhhh - a hexadecimal address. The 'Fence' forms an imaginary boundary across the computer's memory beyond which no single stepping will occur. In this respect it is similar to the optional line number parameter that is used with TRON. A JSR or JMP to a routine on the other (high) side of the fence will cause that code to be executed as normal except that it will not be traced by the DEBUG routine. This is very useful for routines which make

system calls that perform standard functions. For example, while following a piece of code you will probably not want to step through OSWRCH as the function of that routine is well documented - setting fence to &8000 will ensure that no calls into the ROMs are traced, although they will execute correctly. The worked example at the end of this section shows how the command may be used.

Unnn

Set 'Update' to specified value. The parameter is entered in decimal and must lie in the range 0 - 255. This factor determines how often the screen is updated and is measured in executed instructions. Thus, to update the screen after each instruction has been obeyed, 'Update' should be set to 1, which is its default value. A setting of 0 means that no updating will take place, but the code is still executed under the control of DEBUG - this can be useful for following tedious loops where the actual code is not important, but the result and time taken are.

Dnnn

Set 'Delay' to specified value. 'nnn' is entered in decimal and must be in the range 0 - 255. This parameter determines how quickly machine instructions are executed - a value of 0 causes the code to run as fast as possible while 255 will generate a 2 second pause before processing the next instruction.

Bn hhhh

Set breakpoint 'n' to 'hhhh'. 'n' must be either 0, 1 or 2, while 'hhhh' can be any valid address (in hex). The current values of the breakpoints are shown at the section marked 'Breaks'. If, while running a section of code, the program counter reaches a value equal to one of the breakpoints, the beeper will sound and the program will be halted - at this point it can then be continued or abandoned. The breakpoints are 'logical' rather than 'physical' which means that they do not amend any of your code and can, if you wish, exist in ROM. To delete a breakpoint, it is sufficient to type 'Bn' ie. The default setting of 'hhhh' is 0.

Phhhh

Set the program counter. 'hhhh' is the (hex) address of the next instruction to be obeyed.

I

Execute invalid opcode. Quite a number of hex codes are 'invalid' in that they are not part of the 6502's instruction set; these codes are replaced by '???' during disassembly. An example is 'B770' which will load both the Accumulator and the X-register from location &70. If DEBUG tries to execute such an instruction, it will sound the beeper and halt. Pressing ESCAPE will return control to you and will identify the offending instruction. To execute the instruction, type '!'. If you wish to skip the instruction or move to a new piece of code, use the cursor keys to move from it or reset the program counter by means of 'Phhhh'

The disassembly listing at the bottom of the screen is the same as that in MON except that two new fields have been added to the right of the display. Where an instruction refers to any location in memory (even indirectly), the 'Ef.Addr' column will show the ACTUAL location referenced together with the hexadecimal contents of that location. As with the other fields in the DEBUG display, these are updated each time an instruction is executed (assuming 'Update' is set to 1).

NOTES

1) If you enter the routine with a *DEBUG command, each field will be set to its default value, eg. Update = 1, Levels = 99, etc. However, if you enter via a CTRL—D command from HEX or MON, the field values are retained. This is done so that you can return to the routine without losing any parameters you have set up.

2) The 'stack' that is displayed in the last row of the hex section is not the true stack, but a dummy stored in page 10 (&A00 - &AFF). While DEBUG is being used, you should not corrupt this area as you may overwrite stack information and cause your program to trace incorrectly. References to the real stack such as LDA &103,X will be redirected to the dummy stack automatically. (Also, writes to location &FE30 to page in another ROM chip are handled correctly so that **SLAVE** can single step other ROMs).

Similarly, any code stored in page 10 may be corrupted if the DEBUG routine is called and so you should avoid this area if you intend to write a piece of machine code that you may wish to trace.

Example of DEBUG

This simple example shows how to use some of the facilities provided by the *DEBUG command. Unless you are fairly experienced in machine code it is worth working through the example carefully, making sure you key in EXACTLY the right data required.

PROMPT	RESPONSE	REMARKS
>	*DEBUG 900	Enter DEBUG routine and set up default parameters
0900:	CTRL-M	Call Mnemonitor so we can type in some machine code
0900:	LDX #41 TXA	NB '41' is hex
0903:	JSR FFEE INX	
0907:	CPX #5B BCC 902	
090B:	BRK 00 00	The above routine prints out the alphabet
090E:	*GO 900	Try it out !!
<RETURN>	RETURN	
090E:	CTRL-D	Return to DEBUG
090E:	P900	Point to start of the code (Could do this by scrolling with ↑)
0900:	M1 7BC0,X	Select memory to be monitored
0900:	RETURN	The code will be executed and print out the alphabet, halting at 90B
Bleep	ESCAPE	Recover from BRK
090B:	FFFFF	Set Fence so that all subroutines (including OSWRCH) will be traced
090B:	D30	Set Delay so that we can see what's going on
090B:	P900	Return to start of code
0900:	RETURN	Execute code again - notice how the routine will step through the machine's OSWRCH coding and how the stack is used to store three copies of the character that is about to be printed

Notice how the code is terminated with a 'BRK' instruction rather than a 'RTS'. This is because if you run it from DEBUG, there is nowhere to RTS to and the routine will probably start executing some random piece of code, depending on what was on its stack initially.

While the code is running you may ESCAPE at any time and alter registers or DEBUG's own parameters. Pressing ESCAPE once will cause the computer to beep and halt; pressing it again will return control to you, at the input prompt. To resume the code press RETURN when the code has been halted or, to single step, use SHIFT-RETURN.

Once you have mastered this example, try adjusting the DEBUG parameters to see how they affect the way the code is traced.

***CRC fsp | first +len/last**

Perform a cyclic redundancy check on a file or block of memory.

A cyclic redundancy check (CRC) is often used to check the accuracy of a piece of data, especially if it is prone to corruption such as a file on disc or tape. The most common form of corruption is very subtle - for example, two bits being swapped or dropped within a single byte. Normal error detection methods often fail to detect this, which is why CRC's are extensively used. Alteration of a single bit in a section of data will completely change the CRC and it is EXTREMELY unlikely that two different files will have the same CRC.

The **SLAVE *CRC** command uses the same formula as the internal cassette system and which is described in the User Guide.

Output from the routine is a four digit hex number (preceded by '&') which is the CRC for that data. This value acts as a 'tag' or identifier for the data that can be used to test its validity.

Two forms of the command are available:

i) ***CRC** followed by a file specification (in the appropriate format for the filing system currently in use) will open the file, read it and then generate a CRC for it. The only area of memory used in this process is the normal area as used for reading files and by ***EXEC**, so no program in the computer will be lost as a result of a CRC.

ii) ***CRC** followed by an address (in hex) will start checking the contents of ROM or RAM from that point. The number of bytes checked is determined by the '+ len' or 'last' parameter. If the first format is used, '+ len' will be the length of data to be checked, in hex. Otherwise, 'last' will be the very last address used by the CRC routine.

The parameter vet works by checking to see if the first parameter is a valid address - if so, the routine assumes that you wish to CRC an area of memory. If it is not an address (or if it is an incorrectly entered address, eg. b12d) then it is assumed to be a file. With a disc system, this could result in a 'Not found' message if the file doesn't exist. For this reason it is not possible to *CRC a file with the name 'BEEB' as this is also a valid address and will be assumed to be such by the routine.

On very simple use of *CRC is to 'verify' a BASIC program stored on disc or tape is the same as that held in the computer's memory. Suppose a program called 'Program' is stored at &1900, which is the current value of PAGE and is also held on disc. These steps will verify the stored copy:

END — This causes BASIC to recalculate TOP
T% = TOP-1
*CRC 1900 T% — compute a CRC for the program in memory
(**SLAVE** will respond with 'CRC = &E345', say)
*CRC Program
(If valid, **SLAVE** should again give 'CRC = &E345')

***REL first +len/last to**

Relocate machine code.

The section of machine code whose first byte is at address 'first' is relocated to address 'to'. Any memory references to bytes or addresses within the relocated code are suitably adjusted, whereas 'external' references are unchanged. Provided the relocated block consists of pure code without data areas, there is a very good chance that the new code will run straight away in its new location. However, it is possible to refer indirectly to code within the relocated block and no 'relocate' routine can hope to detect this. If the coding is convoluted in this way it will be necessary to check it carefully (using *MON) so that obscure references can be found and changed manually.

As usual, 'first', 'len' and 'last' are entered in hex and define the block of code to be moved (see, for example, the syntax of *MOVE). Similarly, 'to' is the address in hex of the relocated code.

As an example, these instructions will relocate the **SLAVE** 'Print contents of accumulator in hex' routine to run in RAM from location &3000.

```
*CROM SLAVE
```

```
*REL 803F + 15 3000
```

Verify this with:

```
A% = &B7:CALL &3000
```

This should result in 'B7' being output.

Error Messages

ERROR MESSAGES

If **SLAVE** detects an error while it is reading the command line, it generates an error code and prints an appropriate message. These codes can be picked up from the BASIC variable **ERR** and handled by **ON ERROR** coding from within your BASIC program. **REPORT** will repeat the last error message, as usual.

This example shows how BASIC can intercept **SLAVE's** error messages:

```
10 REM PRINT KEY DEFINITION
20 ON ERROR GOTO 70
30 INPUT "'Which key',K%"
40 *EDKEY K%
50 GOTO 30
60
70 IF ERR = 131 PRINT "No such key"
80 IF ERR > 17 GOTO 30
90 REPORT:PRINT " at ";ERL
```

ERR	MESSAGE	REMARKS
128	Bad hex	An illegal character, or an odd number of characters occurred within a hex string delimited by single quotes marks.
129	Bad address	An address was specified which lay outside the range 0 - FFFF.
131	Field out of range	A parameter has been assigned a value which lies outside its acceptable range.
132	Bad number	An illegal digit was found while SLAVE was reading a numeric parameter.
134	Too many bytes	An attempt was made to move more than 32k using '*MOVE'.
136	No such variable	The array passed to *SORT does not exist.
137	Dimension error	An attempt was made to sort more elements than dimensioned in an array, or to sort an array with more than one dimension.
138	String too short	An attempt was made to sort a string shorter than specified by the 'start' parameter. (A common cause of this is to fail to define the zeroth array element).
140	Disc error	SLAVE detected an error while reading a disc.

141	No DFS!	You have attempted to invoke a SLAVE disc routine but the current filing system is not DISC.
142	Not found	An attempt was made to *CRC a non-existent file.
16	Syntax:	This command is not in the correct format.

Summary of Commands

SUMMARY OF COMMANDS

*FX 30	Enable/disable SLAVE and colours	11
Z-BREAK	Enter 'No Language' environment	12
*BAD	Recover 'Bad' program	14
*EXTEND	Extend BASIC program	15
*PACK	Compact BASIC program	16
*SORT	Sort specified array into ascending order	18
*PAGE	Move BASIC program to a new address	19
*ENVELOPE	Display envelope definitions	20
*FIND	Find references to BASIC variable name or keyboard	21
*REPLACE	Replace occurrences of 'name1' by 'name2'	22
*XREF	Cross-reference current BASIC program	23
*TRON	Enable enhanced BASIC trace	24
*TROFF	Disable enhanced trace feature	25
*CHAR	Display character definitions	26
*EDIT	Enter dynamic BASIC editor	27
*MOVE	Move a block of memory	31
*CROM	Choose or inspect sideways ROM	32
*EDKEY	Display function key definitions	35
*SLOW	Slow processor	36
*GET	Find string or bytes in memory	37
*PHEX	Dump memory to printer	38
*GO	Executive machine code	38
*HEX	Enter hex display/editor	41
*MON	Enter the mnemonitor	43
*SPACE	Report free disc space	45
*DGET	Find string or bytes on disc	46
*DISCED	Enter disc sector editor	47
*DIS	Disassemble memory to current output channel(s)	50
*DEBUG	Enter machine code single step/debug routine	53
*CRC	Perform cyclic redundancy check on file or memory	59
*REL	Relocate machine code	61

**Published by: A&F SOFTWARE, Unit 8 Canalside Industrial Estate,
Woodbine Street East, Rochdale.**

© 1985

Phototypesetting & Printing by: C + J BRYNING (Printing) LTD. Rochdale

SLAVE +

ADDENDUM

MICROMAN Computers

Rainford Industrial Estate,
Mill Lane, Rainford,
St Helens, Merseyside WA11 8LS
Telephone: (074488) 5242
V.A.T. Reg No. 374 1479 37

The SLAVE Driver's Guide

Since the introduction of SLAVE a number of changes have been made to the original product, in an effort to cure some minor problems and to implement several suggested improvements. These new features are described below.

*HELP Commands (page 7)

*HELP MCODE will not include the *DGET and *REL commands - these have been replaced by *FORMAT and *VERIFY which are described later. The syntax of some of the original commands has changed slightly and these are given in the relevant sections.

The loss of *REL is not serious as the *DIS command can be used to create source code which can then be relocated; in fact, this system is more versatile.

*FX 30 Command (page 11)

The 'hard' disable feature (*FX 30,128) has been removed. This is no longer necessary due to an extension of the *CROM command.

*PACK (byte) (page 16)

It is now possible to remove redundant colons appearing at the ends of lines by using a switch setting of 16. These may have been left following the use of *PACK 2 to remove REM statements.

*CHAR first (last) (page 26)

The option to print the character set has been removed. Thus, *CHAR must have at least one parameter.

*EDIT (lno) (page 27)

These new features have been included:

- CTRL Holding down the CTRL key together with the cursor up (or down) key will move the editing cursor to the beginning (or end) of the program.
- SHIFT Pressing SHIFT together with cursor left (right) will move the editing cursor to the beginning (end) of the current editing line.
- CTRL-A Adds a line to the program following the edit line. The number of the line so inserted is one greater than that of the current edit line. To insert a block of code into a program the best method is to generate a number of blank lines first, using the CTRL-A facility

cont,

and then to scroll back to type in the text. If this feature causes a line to duplicate an existing line number then the program is automatically renumbered in steps of 10 before the new line is inserted. Thus, no lines are ever lost as a result of using the CTRL-A feature.

The lines generated by this facility initially consist of a line number followed by a space.

- CTRL-G Prompts for and accepts any string. Any editing on the current line is lost, and the cursor is repositioned at the start of the first program line containing an occurrence of the specified string.
- CTRL-K Deletes the current line from the program. Once deleted the line cannot be retrieved.
- CTRL-R Renumbers the program lines from line 10 in increments of 10. Any editing on the current line is lost.
- COPY Moves to the next occurrence of the string found with 'CTRL-G'. If no more occurrences are found the cursor will remain at the start of the current line.

Entering the editor does not affect the function keys in any way and they can be programmed (from BASIC) to produce BASIC keywords or long, or frequently used variable names. A more advanced use is to program the keys to produce control codes, such as those recognised by *EDIT as shown in the following example:

Suppose a program contains the variable P% in a number of places which wish to highlight; we program function key f0 like this:

```
*KEY 0 |||K|AREM The next line contains P%|M||| G
```

Now, type *EDIT and then CTRL-G, entering 'P%' in response to the prompt. When the first line has been located, pressing f0 will prefix it with the REM line and locate the next occurrence. Subsequently pressing f0 will scan the program highlighting all references to P% in the same way!

Here is a breakdown of the function key definition:

```
|||K gives a character 139 - ie. cursor up to previous line
|A same as CTRL-A - insert a line
REM..|M enter new program text
|||G gives a character 135 - ie. press 'COPY'. This gets
the next P%
```

Any of the control keys acceptable to *EDIT can be used in a key definition, eg. |R - renumber, |||I - cursor right, and so on.

*CROM ((+/-)no/rsp) (page 32)

ROMs can now be turned on or off by means of the '+' or '-' prefix to the ROM specification or number.

When the ROM specification is prefixed with a '-' the ROM is disabled and will seem to disappear from the machine. This feature is

useful for preventing clashes with commands that are recognised by more than one sideways ROM. Furthermore, some software will not function correctly if certain ROMs are present in the machine and this command will remove the offending ROMs until they are re-enabled with the '+' option.

Once a ROM has been disabled, it will remain so even after the machine is reset with a soft or hard 'BREAK'. For obvious reasons, you can not recall SLAVE once it has been disabled in this way - the only solution is to switch the machine off and start again.

With the example quoted in the Guide, the command: *CROM -DFS. Will disable the DFS,NET chip. Following 'BREAK' the familiar 'Acorn DFS' message will not appear as the machine is now effectively a tape-only computer - for example, PAGE will be set to &EOO, and all DFS commands will be rejected as the DFS chip is completely disabled (it will not even respond to the *HELP command). To restore the DFS you should type: *CROM +DFS. and then press 'BREAK' so that the chip can be properly initialised.

*PHEX first +len/last (page 38)

This command does not automatically enable the printer so that hex dumps can be displayed on the screen or *SPOOLED to files. If it is required to produce a printed listing, you should issue a CTRL-B (or VDU 2) command to activate the printer. When the printing is complete, disable the printer with CTRL-C (or VDU 3).

*GO (addr) (page 38)

The default setting of 'addr' is taken as &8000 so that the command *GO will enter Slave's 'No-Language' environment.

MEMORY EDITING (page 39)

Both hexadecimal addresses and single hex bytes can be entered in upper or lower case. Thus, '*hex ffee' is a valid SLAVE command.

*HEX (addr)

The status line has been expanded to include both the binary and the BASIC keyword (if applicable) for the current editing address, at all times. The facility to toggle between Binary, Keyword and Mnemonic modes has been removed as mnemonics are available by pressing CTRL-M to enter the Mnemonitor.

*SPACE (drv) (page 45)

This command now also reports the total number of sectors on the discs together with the number of free catalogue entries:

Sectors	061/31E
Max gap	007
Cat space	21

which shows an 80-track disc (31E hex sectors, plus 2 for the catalogue) with 97 (061 hex) free sectors, and space for 21 (decimal)

further files.

***DGET (page 46)**

This command has been deleted as it is now an intrinsic feature of the disc editor.

***DISCED (drv) (page 47)**

The disc editor has been completely redesigned and now operates as follows. The instructions given in the SLAVE Driver's Guide should be disregarded.

The SLAVE disc sector editor allows any sector of a disc to be read, edited and re-written to disc. It will function with any single or double-sided, 40 or 80 track, single density disc in ACORN format. If no drive is specified, the currently selected drive is used. Otherwise 'drv', a number in the range 0 - 3, can be used to select a drive.

The display consists of four separate areas:

- i) The top line displays the current drive, track, sector, and absolute sector, along with the ASCII representation of the byte under the editing cursor.
- ii) The main block of the display is a hex dump of the current sector, eight bytes per line. The offset of each line into the sector is displayed in the far left-hand column and runs from 00 to F8. The start and end of the sector are separated by a dotted line - note that it is not possible to cross this line while scrolling around the sector.

The editing line is located at the centre of the display, and within this line the current editing location is highlighted in yellow. All printable characters are displayed in cyan, while non-printing characters appear in magenta.

- iii) The right-hand side of the screen is taken up with a display of the ASCII equivalents of the bytes printed in cyan in the hex dump.
- iv) The bottom line of the screen contains a prompt (in the form of the byte offset into the sector followed by a colon) after which editing commands may be entered.

The sector editor will accept all the data types described in the Guide under 'Memory Editing' with the exception of address (four character quantities) and OSLI lines (preceded by '*'). It is even possible to assemble directly onto the disk, but note that odd results may occur when calculating branch offsets (the start of the sector will appear to be at &E00).

The editor accepts five explicit three-character commands. These are:

- | | |
|----------|--------------------------------------------------------------------------------------------------------|
| DRV 0..3 | Re-initialises the disc editor and restarts editing at track zero, sector zero on the drive specified. |
| TRK drv | Moves to the track specified (in hex) on the cur- |

rent drive. If any editing has taken place in the sector, confirmation will be required. The range of trk acceptable depends on the size of disc being used and is usually zero to 27 (40 track) or 4F (80 track).

SCT 0..9 Moves to the sector specified on the current drive and track. Again, if any editing has taken place confirmation is required.

ABS sct Moves to the absolute sector specified (in hex) on the current drive. Confirmation may be required. The range of sct acceptable depends on the size of the disc and is usually zero to 18F (40 track) or 31F (80 track).

GET string/'hex' Finds the first occurrence of the string of hex bytes on the disc. The syntax for the string is as described for '*GET'. If a match is found, editing will recommence with the cursor on the first byte of the string. Pressing the COPY key will locate the next match. If no further match can be found on the disc, the search exits at the last sector with the COPY key disabled.

Note that the search commences from the start of the disc, but recommences on COPY from the current editing position. Hence, if it is known that the target string occurs near the end of a disc, it is advisable to press ESCAPE soon after the search begins, move to a point on the disc just before the target string, and press COPY.

The editor recognises the following control keys:

CTRL-U	Deletes the contents of the current command line.
CURSOR LEFT	Moves the editing cursor one byte to the left.
CURSOR RIGHT	Moves the editing cursor one byte to the right.
CURSOR UP	Moves the editing cursor up one line (ie forward eight bytes).
CURSOR DOWN	Moves the editing cursor down one line (ie forward eight bytes).

While holding the SHIFT key:

CURSOR UP	Move to the first line of the sector.
CURSOR DOWN	Move to the last line of the sector.

While holding the CTRL key:

CURSOR LEFT	Moves backwards one sector
CURSOR RIGHT	Moves forwards one sector.
CURSOR UP	Moves forwards one track.
CURSOR DOWN	Moves backwards one track.

If an attempt is made to move off a sector that has been amended, the prompt 'Update disc ?' will be displayed. Pressing 'Y' will update the disc and move, pressing 'N' or any other key will discard the edit and make the move. This does not apply to movement due to 'DRV', 'GET', and COPY.

To use the editor effectively you should be aware of the format of

the information that is stored on the disc, but that is beyond the scope of this manual. The ACORN disc manual provides useful information - especially the layout of the two catalogue sectors, which are probably the two most important sectors on any disc. If the catalogue is severely corrupted, or is not in ACORN format, then SLAVE will probably fail to read the disc and will generate a 'Disc Error' message.

***FORMAT trks (drv)**

Format Diskette.

This command will format a disk to 'trks' number of tracks. The parameter 'drv', if present, specifies the drive on which to format.

If no 'drv' is specified the currently logged drive will be assumed. Before starting, the prompt 'Format : drv (Y/N)?' will be displayed, confirmation of the format then being required. This also serves as a check on the current logged drive if drv was omitted from the command line.

The formatter comprises two passes: format and verify. During each pass the track about to be processing will continue with the failing track being flagged with a '?' suffix.

Pressing 'ESCAPE' at any point will abandon the formatting process: however doing so will leave the first two sectors containing padding characters (hex E5). The DFS will not be able to make sense of this, although 'DISCED' will still read the disc.

The formatter produces ACORN format discs, with the recommended track-track sector staggering for optimum performance regardless of step rate and head load times. Some formatters do not perform this staggering (starting each track with the same sector number) which can result in an increase in access times by a factor of up to two.

***VERIFY (drv)**

Verify Diskette.

This command will read the specified (or default) drive and report the state of the tracks on discs. The track numbers are displayed in decimal, - as with the *FORMAT command - and suspect tracks are indicated by the '?' suffix. The number of tracks on the disc is read from the catalogue and so the verifier can be used with non-standard discs such as those with 35 or 82 tracks.

***DIS (first) (+len/last) (sw) (page 50)**

The number of bytes used and the number of labels contained in the symbol table are no longer reported when performing a 'labelling' disassembly.

***DEBUG (addr) (page 53)**

Here is a more complicated example of the DEBUG command showing how it may be used to trace the progress through another ROM; in this case BASIC.

PROMPT	RESPONSE	REMARKS
	NEW	Clear any BASIC program in store
	*CROM BASIC	Page in BASIC to SLAVE
	*DEBUG 8023	Enter DEBUG at the language initialisation point. If your computer is fitted with BASIC 1 then this address should be 801F.
8023:	FC000 DO	Set fence to ignore the tracing of OS calls
8023:	RETURN	Initialise BASIC (takes about 15s)
	10P.	The '>' prompt indicates that you are BASIC (but running under control of DEBUG!). You will see that the display has halted at a 'JSR &FFF1' instruction. BASIC spends most of its time here, which is the 'Read a line from the keyboard' instruction.
	RETURN	This will cause frenzied activity - the line is entered into the BASIC program at the correct spot and tokenised (ie P. has to be interpreted as 'PRINT')
		The line has been accepted - back to JSR &FFF1
	LIST	Once your command has been translated, the program line is fetched, its number converted from binary to decimal before printing and the rest of the line is decrunched.

Watching BASIC at work in this way is very interesting and the process can be slowed down or single stepped at any time by use of DEBUG's facilities. One thing to avoid is pressing the EACAPE key while in 'BASIC' - this will probably cause a crash as BASIC is never actually paged in (as far as the processing - this will leave you at the DEBUG prompt line at which you can enter changes to Fence, Update and so on.

While running BASIC programs from DEBUG, the screen may scroll when characters are output. This is to be expected and the situation can be recovered by pressing ESCAPE twice while DEBUG is stepping code. As usual the first press will stop processing and the second will restore a new, updated DEBUG screen.

*REL

This command has been deleted.